Management with Condor, Part 3

By Forrest Hoffman

The last two Extreme Linux columns provided an introduction to the Condor workload management system, gave detailed installation and configuration instructions for Beowulf clusters, and showed the details of managing and running MPI jobs (parallel programs that use the Message Passing Interface) with Condor. This month, let's continue looking at Condor, explore some of its advanced features, and check out its powerful queuing capabilities for lots of serial tasks.

First, let's quickly review Condor's basic characteristics. Condor was designed to make effective use of a pool of computing resources, whether they're dedicated nodes in a cluster or disparate workstations distributed across a network. A Condor *pool* consists of a *central manager* and a number of other machines that join the pool as participating resources. Condor provides a job queuing mechanism, scheduling policy, job and user priority schemes, and resource classification mechanisms. It matches job requirements (*job ClassAds*) with advertised resource attributes (*machine ClassAds*) to decide where and how jobs should be executed.

Condor jobs run in one of a number of *universes*, where each universe has different characteristics. The `vanilla` universe may be used to run any serial task. The `standard` universe supports serial tasks, but provides automatic process migration among nodes and remote system calls (back) to the originating host. However, programs run in the `standard` universe must be linked with Condor's own libraries, and may not perform some kinds of operations.

The `PVM` and `MPI` universes provide support for parallel programs, while the `globus` universe allows users to submit Globus (grid computing) jobs. Details on configuring and using the `MPI` universe were presented in last month's column (available online at http://www.linuxmagazine.com/2003-11/extreme_01.html).

The `java` universe runs programs written for the Java Virtual Machine (JVM), and the `scheduler` universe is used internally for immediate job execution.

Specific instructions for installing Condor on a typical Beowulf cluster were provided in the October column (available online at http://www.linuxmagazine.com/2003-10/extreme_01.html). The rules under which jobs may be run on these different resources can be configured for each machine.

For instance, a researcher may be glad to have jobs run on her workstation between the hours of 5:00 p.m. and 8:00 a.m. as long as no keyboard activity is present. As a result, Condor can effectively harvest cycles from computers that are not actively running other tasks.

Condor provides mechanisms for both job and user priorities. Job priorities allow the user to control the order of execution of submitted jobs. User priorities, established by the system administrator, control the distribution of compute cycles among jobs owned by various users. Condor adjusts each user's effective (run-time) priority to ensure that everyone gets his fair share allocation of cycles. Jobs run in the `standard` universe may even be preempted (checkpointed and halted), thereby freeing up machines for a higher priority user's jobs.

Jobs are submitted to Condor using *condor_submit* and a submit description file. This file contains all the information Condor needs to run the job, including the name of the executable to run, command-line arguments to the program, run-time directories, input and output files, and job-related requirements used in creating the job ClassAd.

Basic submit description files were discussed in previous columns. Here, let's see how to use some more advanced options.

Running Lots of Jobs in the `vanilla` Universe

In addition to running parallel programs, Beowulf clusters are often called upon to run "distributed" jobs or lots of seri-

FIGURE ONE: A Condor submit description file for running the same program in multiple directories

```
####################################
# Analyze lots of input.dat files in
# different directories
####################################
Executable = analyze
Universe   = vanilla
input      = input.dat
output     = output.dat
error      = error.dat
log        = analyze.log

Initialdir = run0
Queue

Initialdir = run1
Queue

Initialdir = run2
Queue

Initialdir = run3
Queue
```

FIGURE TWO: A Condor submit description file for running the same program in multiple directories using the `$(Process)` macro

```
####################################
# Analyze lots of input.dat files in
# different directories using macros
####################################
Executable = analyze
Universe   = vanilla
input      = input.dat
output     = output.dat
error      = error.dat
log        = analyze.log

Initialdir = run$(Process)
Queue 4
```

al tasks simultaneously. Such jobs may involve independent analyses of separate images, rendering of individual movie frames, many gene sequence comparisons, and Monte Carlo techniques, among others. These are all tasks that either work independently on a single data set or perform the same operations or independent analyses on separate data sets. Condor was designed to make this sort of distributed or "massively serial" computing easy.

For example, if a single program needs to be run against a number of different datasets and that program creates or uses files with fixed file names, it may be desirable to perform these runs in separate directories to avoid conflicts if the jobs all run simultaneously.

Figure One contains a submit description file that demonstrates this capability. The same executable program (*analyze*) runs four times, once in each directory, *run1*, *run2*, *run3*, and run4. The user initially creates the *input.dat* file in each directory before submitting the description file to Condor. After the file is submitted using *condor_submit*, all four jobs are queued and run simultaneously if adequate resources are available in the pool. An *output.dat*, *error.dat*, and *analyze.log* file is created in each directory, along with any other files that the analyze program may create.

Alternatively, using pre-defined macros can make the submit description file even shorter. The file shown in *Figure Two* does the same thing as the one shown in *Figure One:* it starts four jobs, one in each run directory. The `$(Process)` macro expands to be the process number (beginning at zero) for each task. The `Queue 4` line creates four jobs or tasks that can run simultaneously.

Similarly, if a couple hundred images need to be processed, the jobs can all be submitted using a single submit description file like that shown in *Figure Three*. This file creates two hundred jobs that run *image_proc* with slightly different command line arguments. For this example, the *image*N*.png* files are inputs (where *N* goes from 0 to 199), and the results are output to resultsN. Separate output (*stdout*), error (*stderr*), and log files are generated for each process.

Of course, running two hundred jobs means that you'll receive two hundred email messages notifying you of each job completion. To disable email notifications, add the line `notification = never` to the submit description file.

Compiling Codes for the Standard Universe

In the `standard` universe, Condor provides checkpointing (generation of a file containing a snapshot image of the current state of the job) and remote system calls. If a job must be stopped or if the machine on which it was running crashes, Condor uses the last checkpoint image to restart the job (from where it left off) on another machine.

Remote system calls make a job appear to execute on its home machine. A process called condor_shadow runs on the machine where the job was submitted, and performs the needed system calls and I/O operations even though the job may actually be running on a different machine. The relevant data are then forwarded to the job process wherever it is running.

To take advantage of these features and run in the `standard` universe, programs must be relinked with the Condor libraries using *condor_compile*. This is usually accomplished by putting *condor_compile* in front of your usual link or build command. For example, to build a new binary from an existing object file, the syntax looks like:

```
% condor_compile gcc -o analyze analyze.o
  functions.o util.o
```

Since you need the unlinked object files, it may not be possible to relink commercial software or programs for which

FIGURE THREE: A Condor submit description file for running the same program with different command line arguments

```
####################################
# Process 200 images and save the
# results in separate files
####################################
Executable = image_proc
Universe   = vanilla
output     = image_proc.out.$(Process)
error      = image_proc.err.$(Process)
log        = image_proc.log.$(Process)

Arguments  = -w 640 -h 480 -i
image$(Process).png -o result$(Process)
Queue 200
```

you have only the executable file. The *condor_compile* script can also be used in compile and link commands…

```
% condor_compile gcc -O -o time_waster
  time_waster.c -lm
```

… or even in front of *make* or a *build* script:

```
% condor_compile make
% condor_compile build
```

Once a program is linked in this fashion, it can be used as an executable in a submit description file to be run in the `standard` universe. In addition, it can be run interactively for standalone checkpointing. With standalone checkpointing, a code can run and then be forced to write a checkpoint image and stop. The job can then be restarted manually later using the checkpoint image and will continue where it left off.

For example, in *Figure Four*, a program called time_waster is compiled using *condor_compile* then executed from the command line. The Condor libraries notify the user that the code will checkpoint to a file called time_waster.ckpt, and that remote system calls are disabled. After pass nine, say, the user sends the process a `SIGTSTP` signal by pressing CONTROL-Z, after which the program writes a checkpoint image to *time_waster.ckpt* and exits. The user then restarts the program manually on a different machine (pointing the code to the correct checkpoint image file), and it continues right where it left off in pass 10. Sending a `SIGUSR2` signal to the process causes it to checkpoint and continue running.

FIGURE FOUR: Standalone checkpointing and restarting

```
[forrest@node001 zim]$ condor_compile gcc -O
  -o time_waster time_waster.c -lm

[forrest@node001 zim]$ ./time_waster
Condor: Notice: Will checkpoint to
  ./time_waster.ckpt
Condor: Notice: Remote system calls disabled.
After pass 0, y = 3.09806
After pass 1, y = 6
After pass 2, y = 8.79703
After pass 3, y = 11.5229
After pass 4, y = 14.195
After pass 5, y = 16.8239
After pass 6, y = 19.4168
After pass 7, y = 21.9788
After pass 8, y = 24.5137
After pass 9, y = 27.0246
User defined signal 2

[forrest@node001 zim]$ slogin node002
[forrest@node002 forrest]$ cd zim
[forrest@node002 zim]$ ./time_waster
  -_condor_restart time_waster.ckpt
Condor: Notice: Will restart from
  time_waster.ckpt
After pass 10, y = 29.5137
After pass 11, y = 31.9832
After pass 12, y = 34.4346
After pass 13, y = 36.8693
After pass 14, y = 39.2884
       .
       .
       .
```

FIGURE FIVE: A submit description file for *time_waster* to run eight times in the `standard` universe

```
####################################
# Waste lots of time
####################################
Executable   = time_waster
Universe     = standard
output       = time_waster.out.$(Process)
error        = time_waster.err.$(Process)
log          = time_waster.log.$(Process)
notification = never

Arguments  = -p 10
Queue 8
```

Checkpointing and restarting (along with remote system calls) allows the Condor system to preempt jobs and later migrate them to other machines (of the same architecture) where they continue where they left off.

This program can be scheduled to run under the Condor system by creating an appropriate submit description file — like that shown in *Figure Five* — which references the `standard` universe. This file queues up eight identical *time_waster* jobs.

Once submitted using *condor_submit*, as shown in *Figure Six*, *condor_q* shows that all eight jobs are running. Using the *condor_vacate* command, you can force the jobs running on some machines, *node002* and *node003* in this example, to be stopped and removed from those machines. (It's not normally necessary to ever use the *condor_vacate* command explicitly since Condor normally handles these details when necessary. It's used here merely to demonstrate the capability.)

`standard` universe jobs that are preempted in this fashion checkpoint and exit so they can be restarted automatically at a later time. Jobs running in the `vanilla` universe that are preempted are killed and are restarted from the beginning by Condor at a later time.

In *Figure Six* we see from the *condor_status* listing that jobs are being preempted on the affected hosts (where `State` is `Preempting` and `Activity` is `Vacating`). A check of the

See Extreme, *pg. 59*

FIGURE SIX

```
[forrest@node001 zim]$ condor_submit time_waster.condor
Submitting job(s).........
Logging submit event(s).........
8 job(s) submitted to cluster 56.

[forrest@node001 zim]$ condor_q
-- Submitter: node001.cluster.ornl.gov : <192.168.54.1:54461> :
node001.cluster.ornl.gov
ID           OWNER         SUBMITTED       RUN_TIME      ST       PRI      SIZE         CMD
56.0         forrest       10/11 01:00     0+00:02:52    R        0        3.4          time_waster -p 10
56.1         forrest       10/11 01:00     0+00:02:49    R        0        3.4          time_waster -p 10
56.2         forrest       10/11 01:00     0+00:02:45    R        0        3.4          time_waster -p 10
56.3         forrest       10/11 01:00     0+00:02:43    R        0        3.4          time_waster -p 10
56.4         forrest       10/11 01:00     0+00:02:41    R        0        3.4          time_waster -p 10
56.5         forrest       10/11 01:00     0+00:02:39    R        0        3.4          time_waster -p 10
56.6         forrest       10/11 01:00     0+00:02:47    R        0        3.4          time_waster -p 10
56.7         forrest       10/11 01:00     0+00:02:37    R        0        3.4          time_waster -p 10

8 jobs; 0 idle, 8 running, 0 held
[forrest@node001 zim]$ condor_vacate node002 node003
[forrest@node001 zim]$ condor_status

Name              OpSys        Arch         State         Activity     LoadAv     Mem       ActvtyTime
vm1@node001.c     LINUX        INTEL        Owner         Idle         1.000      1008      25+01:51:51
vm2@node001.c     LINUX        INTEL        Unclaimed     Idle         0.010      1008      0+00:10:05
vm1@node002       LINUX        INTEL        Preempting    Vacating     1.980      503       0+00:00:04
vm2@node002       LINUX        INTEL        Preempting    Vacating     2.090      503       0+00:00:05
vm1@node003       LINUX        INTEL        Preempting    Vacating     1.940      503       0+00:00:04
vm2@node003       LINUX        INTEL        Preempting    Vacating     2.050      503       0+00:00:05
vm1@node004       LINUX        INTEL        Claimed       Busy         1.990      503       0+00:07:30
vm2@node004       LINUX        INTEL        Unclaimed     Idle         1.000      503       0+00:41:47
vm1@node005       LINUX        INTEL        Unclaimed     Idle         1.000      503       0+00:12:59
vm2@node005       LINUX        INTEL        Unclaimed     Idle         1.000      503       0+00:13:11
vm1@node006       LINUX        INTEL        Claimed       Busy         1.240      503       0+00:07:35
vm2@node006       LINUX        INTEL        Claimed       Busy         1.750      503       0+00:07:33
vm1@node007       LINUX        INTEL        Unclaimed     Idle         1.000      503       0+00:40:19
vm2@node007       LINUX        INTEL        Unclaimed     Idle         0.990      503       0+00:40:05
vm1@node008       LINUX        INTEL        Claimed       Busy         1.990      503       0+00:07:41
vm2@node008       LINUX        INTEL        Unclaimed     Idle         1.000      503       0+01:00:08
      .
      .
      .

                  Machines     Owner        Claimed         Unclaimed      Matched      Preempting
INTEL/LINUX       128          1            4               119            0            4
Total             128          1            4               119            0            4
[forrest@node001 zim]$ condor_q

-- Submitter: node001.cluster.ornl.gov : <192.168.54.1:54461> :
node001.cluster.ornl.gov
ID           OWNER         SUBMITTED       RUN_TIME      ST       PRI      SIZE         CMD
56.0         forrest       10/11 01:00     0+00:07:55    I        0        4.1          time_waster -p 10
56.1         forrest       10/11 01:00     0+00:07:51    I        0        4.1          time_waster -p 10
56.2         forrest       10/11 01:00     0+00:07:45    I        0        4.1          time_waster -p 10
56.3         forrest       10/11 01:00     0+00:07:43    I        0        4.1          time_waster -p 10
56.4         forrest       10/11 01:00     0+00:09:26    R        0        3.4          time_waster -p 10
56.5         forrest       10/11 01:00     0+00:09:24    R        0        3.4          time_waster -p 10
56.6         forrest       10/11 01:00     0+00:09:32    R        0        3.4          time_waster -p 10
56.7         forrest       10/11 01:00     0+00:09:22    R        0        3.4          time_waster -p 10

8 jobs; 4 idle, 4 running, 0 held
```

Extreme, *from pg. 36*

job queue using *condor_q* shows that four jobs that were preempted are idle and the remaining four are running. Some time later, Condor restarted the four preempted jobs, and they continued from where they left off.

Requirements and Rank Expressions

Two additional keywords are frequently used in submit description files: `requirements` and `rank`. These powerful commands allow you to specify exact ClassAd expressions for submitted jobs that would otherwise be set to default values by *condor_submit*. ClassAd attributes are case sensitive, so great care must be taken when specifying `requirements` and `rank` values.

Valid ClassAd attributes are those that appear in a machine or a job ClassAd. To see all the machine ClassAd attributes, run `condor_status -l` for all machines in a pool, or `condor_status -l hostname` for a specific hostname. The Condor documentation lists all the valid attributes that may be specified. The most commonly used job ClassAd attributes are those that refer to memory requirements, floating point performance, computer architecture, and operating systems.

For example, in a heterogeneous pool of machines, a job can be ensured to run only on Intel boxes running Linux if the following line is added to the submit description file:

```
Requirements = (Arch == "INTEL" &&
  OpSys == "LINUX")
```

Since these attributes can be used as macros, it's fairly easy to write a submit description file that starts the correct executable on the correct architecture and operating system. For example:

```
Requirements = ( (Arch == "INTEL" && OpSys
  == "LINUX") ||
  (Arch == "SGI" && OpSys == "IRIX65") )
Executable = analyze.$$(OpSys).$$(Arch)
```

The `rank` expression affects the numerical preference value for machines during match making. For a job that needs to run on the machine with the most memory, you might use:

```
Rank = memory
```

To be more explicit, you might want your job to require at least 32 MB of memory, but prefer to run on machines with 64 MB or larger. You can express that with:

```
Requirements = Memory >= 32
Rank = Memory >= 64
```

For a job that needs the best floating point performance, you can say:

```
Rank = kflops
```

Preference for a particular list of machines can also be specified:

```
Rank = ( (machine ==
"node007.cluster.ornl.gov") ||
  (machine == "node008.cluster.ornl.gov")
)
```

And preferences for these machines can be further specified using:

```
Rank = ( ((machine ==
  "node007.cluster.ornl.gov")*10) +
  (machine == "node008.cluster.ornl.gov")
```

so that `node007` is preferred 10 times over `node008`, but both are preferred over all others in the pool.

Condor is a valuable asset for managing big computational workloads

As you can see, Condor is an extremely powerful and flexible system. It is particularly good for managing lots of serial jobs, for harvesting cycles from disparate workstations, and for scheduling distributed and parallel tasks in a Beowulf cluster environment. It has capabilities and features not presented here, including the ability to define dependencies among submitted jobs, and support for grid computing using a mechanism called "flocking" or through the Globus toolkit. Even a graphical interface, called CondorView, is available for looking at machine and job statistics.

Even if Condor is not the best system for managing a wide variety of parallel jobs using different versions of MPI on your cluster, it's worth watching. The developers are planning to add support for more recent MPICH releases, as well as for LAM/MPI.

With features not available in many other batch systems, Condor is a valuable asset for managing big computational workloads in many computing environments.

Forrest Hoffman is a computer modeling and simulation researcher at Oak Ridge National Laboratory. He can be reached at forrest@climate.ornl.gov.